Python Introduction with Math

Main Points

1 Programming and Mechanical Engineering Technology. . . What?

Mechanical Engineers and, by extension, Mechanical Engineering Technologists cannot expect to only work with mechanical-only devices in the modern world. Almost every traditional mechanical system is somehow connected to a computer or other programmable device. There are many reasons for learning at least some computer programming; a few of which are listed below.

- Reinforce logical and systematic problem solving methods
- Improve creative problem solving and trouble-shooting skills
- Not everything can (nor should) be done with *Excel*
- Work more efficiently by automating boring and tedious workflows
- Create or prototype custom, industry-specific software solutions
- Communicate more effectively with professional programmers
- Read the code for machines/devices to aid in understanding their operation
- Better understand and utilize programmable micro-controllers and Internet of Things devices
- Expand career opportunities outside of your major
- Ever hear of mechatronics or *Industry 4.0*?

2 Why *Python*?

- Considered to be a very readable and beginner-friendly programming language (*Python's* original author “famously” stated that code is read more often that it is written)
- Interpreted (no compiling required) resulting in shorter development times
- Free and easy to install
- Can be extended by many, many external libraries and modules
    - Numeric *Python* (*NumPy*) for numerical anaysis
    - *Matplotlib* for high quality plots
    - *Pandas* for data science
    - Machine learning and artificial intelligence (AI) libraries
- Ported to micro-controller devices via *CircuitPython* and *MicroPython*

3 A 10,000 Foot View of a *Python* Script

The following block shows the code from a sample *Python* script in the upper portion of the block and the results from executing the script in the lower portion.

A Sample Script

```
 1 # sample_script.py
 2 
 3 values = [4, 9, 3, 0, 7]
 4 print("A review of", values)
 5 for number in values:
 6     if number == 0:
 7         print(f"  This value is zero")
 8     elif not number % 2:
 9         print(f"  {number} is even")
10     else:
11         print(f"  {number} is odd")
12 
```

```
A review of [4, 9, 3, 0, 7]
  4 is even
  9 is odd
  3 is odd
  This value is zero
  7 is odd
```

- Line 1 is a comment and ignored by *Python*
- Line 3 creates a list of 5 numbers and assigns the list to the name `values`
- Line 4 displays a statement about the list `values`
- Line 5 initializes a `for` loop for each item in `values`, using the name `number` for the item each time through the loop Thecolon `:` at the end of line 5 is necessary

- Lines 6, 8, and 10 are indented by 4 spaces because they belong to the `for` loop code block
- Lines 7, 9, and 11 are indented by an additional 4 spaces because they each belong to the `if`, `elif`, and `else` commands (respectively) `The`colons at the end of the `if`, `elif`, and `else` commands are required
- Lines 6-11 will be executed 5 times; once for each of the items in the list named `values`
- The `print()` functions on lines 7, 9, and 11 display (print) results statements to the screen
- The `print()` function on line 7 only prints if the value of `number` is equal to zero
- The `print()` function on line 9 only prints if `number` is not zero and is divisible by 2
- The `print()` function on line 11 prints for all other conditions; when `number` is not equal to zero and not divisible by 2
- The `print()` functions on lines 9 and 11 utilize *f-strings* to include the value of `number` in the printed statements
- The blank line at the end (line 12) is intentional and a good practice

4 Executing *Python* Commands

Python commands and expressions can be executed without needing to create scripts. Doing so allows *Python* to be used as a calculator or to test bits of code.

Starting *Python* to get a REPL prompt

```
% python
Python 3.9.13 (main, Oct 13 2022, 16:12:19)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- From a *Python* interpreter command prompt
  - This environment is also known as the REPL (Read, Evaluate, Print, and Loop)
  - The prompt usually consists of three "greater than" arrows, like so >>>
  - Typed commands are executed when the `[return]` key is pressed

Entering commands/expressions at a *Python* prompt

```
>>> print("Hello")
Hello
>>> x = 21
>>> print(x * 2)
42
>>> 2 + 3
5
>>> (3**2 + 4**2)**0.5
5.0
```

- From a code block in a *Jupyter* notebook
  - Execute a single command by pressing `[shift]-[enter]` after typing it
  - Type multiple lines by pressing `[return]` after each line then press `[shift]-[enter]` to execute all of the lines

5 Editing and Running Scripts

Python scripts are similar to computer applications or programs that you run everyday. One of the biggest differences is that scripts require *Python* to be installed so it can execute the code in the script. At their most basic, scripts are lines of commands, functions, and expressions that are executed in order from the top down.

5.1 Editing Scripts

- *Python* scripts must be written in plain text
  - Good - a stand-alone text editing applications
  - Better - an editor built into a *Python* capable integrated development environment (IDE)
- Word processors like *Word* should never be used for writing or editing scripts
- *Python* installations have a built-in text editor/IDE named *IDLE*
- Dedicated coding text editors, like *Visual Studio Code* (*VS Code*), often add nice features; including the ability to run scripts directly from the editor
- Good, beginner-friendly IDEs
  - *Thonny* - can install and use a "local" version of *Python* from within the application
  - *Mu* - possibly the best IDE for *CircuitPython*

5.2 Naming scripts

- No spaces in the name
- Must start with a letter (or an underscore character, but this is infrequent)
- May include only letters, numbers, and underscores
- Should be descriptive of what the script is used for
- Must end with `.py` file extension
- Some valid script names...
  - `stress_calculator.py`
  - `fibonacci.py`
  - `_calculator.py`
  - `easy_as_123.py`
  - `uscs2si.py`

5.3 Running Scripts

- From a console command line prompt such as Windows Power Shell or the MacOS Terminal app
  - Type `python` or `python3` followed by the script name, including the `.py` extension
  - Examples...
    - `python hello.py`
    - `python3 hello.py`

Running a script from console prompt

```
% python hello.py
Hello, World!
```

- From a *Python* prompt
  - Change to the directory containing the script file before starting *Python*
  - Start *Python*, type `import script_name` at the >>> prompt, and press `[return]`

Running a script from a *Python* REPL prompt

```
>>> import hello
Hello, World!
```

6 *Jupyter* Notebooks and *Python*

Jupyter notebooks are files that can incorporate blocks of executable *Pytyon* (and some other languages) code, formatted text, images, and more. They are great for prototyping scripts, trying out code snippets, or documenting engineering problems that include some amount of programming. They usually run locally via a web browser using the default version of *Python* installed on the computer. Completely web-based implementations of *Jupyter* notebooks are available.

6.1 Cell Types

Standard *Jupyter* notebooks can contain a mix of three possible types of cells. Many online notebook interfaces that are based on *Jupyter* may only contain Markdown and Code cells.

- Code - executable *Python* (or other languages) code
- Markdown - formatted text, formulas, and images
- Raw - non-executable code-like text

6.1.1 Executing cells

- All cell types can be executed
  - Code cells execute the *Python* commands
  - *Markdown* cells generate the formatted text
  - Raw cells leave the text “as-is”
- `[shift]+[return]` is used to execute a cell and jump to the next cell or start a new cell
- `[control]+[return]` on Windows (or `[command]+[return]` on a Mac) is used to execute a cell and keep the current cell selected

6.1.2 Operational Modes

- Edit mode
  - Green border (usually)
  - Text cursor
  - Type commands or text in cells
  - Must be in edit mode to type or edit *Python* commands
  - Double-clicking inside a cell will enter edit mode
  - Pressing `[return]` while in command mode with a cell highlighted will enter edit mode

- Command mode
  - Blue border (usually)
  - Standard arrow-style pointer
  - Used to manage the notebook's structure and appearance
  - Copy, paste, and delete cells
  - Click outside of a text editing area (left or right of the cell) to enter command mode
  - Execute a cell that is being edited to enter command mode

6.1.3 Common command mode shortcuts

- A: add a new cell above the selected cell
- B: add a new cell below the selected cell
- X: cut the selected cell
- M: change the selected cell to *Markdown*
- Y: change the selected cell to Code
- R: change the selected cell to Raw

6.2 Code Cell Numbering

- `In [ ]:` - input cell before being executed
- `In [1]:` - input cell after being executed
- `Out[1]:` - output cell belonging to `In [1]:`
- Certain output cells do not get numbers; for example, output created by printing or plotting
- Cell numbers are not re-used; they keep counting up until the notebook or its kernel is restarted
- The last state of all input and output cells is maintained when a notebook is saved
- Values assigned to variables are not remembered after a kernel or notebook restart

6.3 Editing Formatted Text (*Markdown*) and Raw Text Cells

Markdown cells are good for including instructions, descriptions, explanations, and general commentary along with your code. They can also be used for linking and displaying image files that are located in the same directory as the notebook file. Lastly, *Markdown* cells can include formulas/equations. Raw cells' primary purpose is to display code or code-like text.

6.3.1 Common *Markdown* symbols used for formatting text

- Include * immediately before and after text for *italic* text, i.e. `*italics*`
- Include ** immediately before and after text for **bold** text, i.e. `**bold**`
- Place a $ before and after special code to generate inline mathematical formulas using LaTeX; for example `$\beta = \frac{\pi}{4}$` results in $\beta = \frac{\pi}{4}$
- Use - or * followed by a space at the beginning of a line to create a bullet
- Use # followed by a space at the beginning of a line to make a title (first order heading)
- Use ## followed by a space at the beginning of a line to make a second order heading
- Use ### (or more) followed by a space at the beginning of a line for a third (or higher) order heading
- Place ___ (3 underscore characters) at the beginning of a line to create a horizontal dividing line

6.3.2 Raw cells

- Used less frequently
- Mainly used to show sample code
- Not executable

6.4 Running Scripts in *Jupyter* Notebooks

- Place the script file in same directory as the *Jupyter* notebook
- Execute `run script.py` in a code cell using the actual script name, i.e. `run hello.py`

```
In[ ]: run hello.py
```

7 Spaces and Capitalization Are Important to *Python*

- Spaces between values, variable names, and operators are acceptable and ignored
- Unexpected spaces at the beginning of lines will generate errors
- Indentation spaces at the beginning of lines are used to designate blocks of code
- Indentation must be consistent when used
- Variable names with the same spelling but have differing letter cases are not the same; i.e. `force`, `Force`, and `FORCE` are different names
- All *Python* commands and functions use lower case characters only
- *Python* PEP 8 gives general guidelines for code style (https://peps.python.org/pep-0008/)

Spaces between values and operators are ignored

```
>>> x = 2
>>> x*2
4
>>> x * 2
4
```

```
>>> x     *      2
4
>>> print ( "Spaces" )
Spaces
```

Unexpected spaces at the beginning of a line cause errors

```
>>> x = 2
>>>  x * 3
  File "<stdin>", line 1
    x * 3
    ^
IndentationError: unexpected indent
```

Indentation spaces are required to designate blocks of code

```
for letter in "hello":
    letter_upper = letter.upper()
    print(letter_upper)
```

```
H
E
L
L
O
```

Upper and lower case are not the same to *Python*

```
>>> a = 5
>>> A = 6
>>> print(a)
5
>>> print(A)
6

>>> # the next line will cause an error due to the capital P in Print
>>> Print(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Print' is not defined
```

8 Including Comments in Code

It is a good idea and common practice to include comments in your code to describe "why" and not "how" something was done. Lines of code can also be turned into comments to keep them from executing. This is very helpful while troubleshooting (aka debugging) a script.

- Use the # symbol to indicate the start of a comment
- Anything after # is ignored (not executed) by *Python*
- Lines can start a with the # symbol; causing the entire line to be ignored
- Comments can be placed to the right of *Python* commands
- If a # is in a string it will not create a comment
- A common practice for *Python* scripts is to enclose multiple lines at the top of a script within triple quotes to form a single large comment (see below) called a docstring-style comment

Python comments

```
"""
This is a docstring-style comment
This program does such and such
It was written by Brian Brady
Last revision: 11/20/2022
"""

# This is a single-line comment
x = 42    # Answer to the universe and everything
# y = x - 30
print(x)
```

9 Three Important *Python* Object Types

Nearly everything in *Python* is an **object**. *Python* has three object types for values that are used very frequently; integers, floating point values (floats), and strings. Use `type(x)` to check the type of an object, i.e. `type(42)` or `type("Hello")`. Only integers and floats can be used in math expressions.

- Integers
  - Whole numbers with no decimal part; Values such as . . . , -3, -2, -1, 0, 1, 2, 3, . . .
  - **`int`** type
  - Max size varies based on system memory, but they can be very large

Maximum integer size

```
>>> import sys
>>> print(f"{sys.maxsize:,d}")
9,223,372,036,854,775,807
```

- Floating point values, aka floats
  - Non-integer numeric values and integers with a decimal point
  - Values such as 2.0, 2., and -3.45
  - **`float`** type
  - Floats can be very large in *Python*

Maximum integer size

```
>>> print(f"{sys.float_info.max:.16E}")
1.7976931348623157E+308
```

- Strings
  - Any type of character or characters contained within quotes
  - **`str`** type
  - Double or single quotation symbols can be used...
    - `"Hello"`
    - `'Python'`
  - Numbers within quotes are strings
    - `"42"`
    - `'2.78'`

Determining object types

```
>>> type(3.14)
<class 'float'>

>>> type(2.)
<class 'float'>

>>> type(42)
<class 'int'>

>>> type("Python")
<class 'str'>

>>> type("3.14159")
<class 'str'>
```

Each of the three object types previously mentioned may be converted to one of the other types. Examples for each of the following are included in code cells.

- To an integer...
  - Use the `int()` function
  - Convert floats - the decimal point and all numbers to the right are dropped
  - Convert strings that contain only an integer value
  - Cannot directly convert a string containing a float to an integer

Converting to integers

```
>>> int(3.14159)
3
>>> int("42")
42
>>> int(float("3.14159"))
3
```

- To a float...
  - Use the `float()` function
  - Convert integers - adds a decimal point and zero
  - Convert strings that contain only a float or an integer value

Converting to floats

```
>>> float(42)
42.0
>>> float("3.14159")
3.14159
```

- To a string...
  - Use the `str()` function
  - Convert integers or floats - numeric values are simply enclosed in quotes

COnverting to integers

```
>>> str(42)
'42'
>>> str(3.14159)
'3.14159'
```

10 Using Names, aka Variables

A great deal of the flexibility and power gained from using a programming language to automate tasks comes from the using variables, also called names in the *Python* world.

- Almost any value or object can be assigned to a name (variable)
- Assign a value to a name like so; `variable = value`
- Assing the results of an expression or function similarly; `variable = expression`
- Variables can be used in place of the value or object after assignment
- Names in *Python* are case-sensitive, i.e. `angle` and `Angle` are not the same
- Variable naming rules...
  - Must start with a letter or underscore
  - Can also contain numbers and underscores
  - No spaces and other special characters
  - Restricted *Python* names (keywords) cannot be used as variable names
  - Don't use built-in function or command names as variable names
    - Using function names will overwrite the functions and they will not work
    - Trying to use most command names will generate an error
    - Example: using `abs = 100` will keep the `abs()` function from working
- Using `del variable_name` will clear `variable_name`
- Try to use descriptive variable names whenever possible
- It is preferred to use underscores between words in long names instead of capitalizing the second word, i.e. `max_pressure` not `maxPressure`

Restricted names (keywords) that cannot be used as variables

```
>>> help("keywords")

Here is a list of the Python keywords.  Enter any keyword to get more
 ↪  help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not
```

Assigning descriptive names (variables)

```
>>> answer_to_universe = 42
>>> print(answer_to_universe)
42
>>> radius = 10
>>> print('radius =', radius)
radius = 10
>>> bad_guy = "Darth Vader"
>>> total_time = 30 * 60
>>> total_time
1800
```

- *Python* does not display any results when a value or calculation is assigned to a name (see above). Use the `print()` function in order to explicitly display such results

11 Displaying Output Using `print()`

When working from a *Python* prompt or within a *Jupyter* notebook most results will be displayed implicitly (without the need to do anything special). However, within scripts the use of the `print()` function is required to explicitly display output. The `print()` function always provides more control of the displayed output (more on this at another time).

- Use the `print()` function to explicitly display results from a prompt or script
- The `print()` function displays the results of (nearly) anything included in the parentheses
- Multiple items/objects can be displayed with a single `print()` command
    - Items must be separated by commas within `print()`
    - A space is added between each item that was separated by a comma
- Multiple strings in a `print()` function can be connected together (concatenated) by separating them with the + symbol

Implicit “printing” from the REPL

```
>>> 2 + 2
4
>>> "Hello, World!"
'Hello, World!'
>>> abs(-100)
100
```

Explicit printing using print()

```
>>> print("Python is awesome")            # a single string
Python is awesome

>>> print("Hello " + "again")             # printing concatenated strings
Hello again

>>> print('Lumberjacks', "Parrots", 42)   # multiple items with commas
Lumberjacks Parrots 42

>>> print(4 * 5)                          # calculated then printed
20

>>> print(2 * 'Hello')                    # repeat strings by multiplying
HelloHello
```

12 A Preview of *f-Strings*

Formatted string literals, aka *f-strings* were added to *Python* in version 3.6. They allow strings to include values from expressions or variables with specific formatting if desired. This topic will be covered in more depth in another document.

- Add the letter f immediately before the opening quote for a string
- Include sets of curly braces {} in the string in places a variable value or expression result is desired
- Include a variable name or expression in each curly brace pair
- Curly braces act like blanks in a "fill-in-the-blank" statement

Examples of *f-string* use

```
a = 2
b = 3
x = 42
first_name = "Brian"
last_name = "Brady"
print(f"22/7 = {22/7} and is close to pi")
print(f"The sum of {a} and {b} is {a + b}")
print(f"The answer to the universe is {x}")
print(f"My name is {first_name} {last_name}")
```

```
22/7 = 3.142857142857143 and is close to pi
The sum of 2 and 3 is 5
The answer to the universe is 42
My name is Brian Brady
```

13 *Python* Modules

There are many, many modules of specialized functions available for use in *Python*. Many of the modules are included with the standard *Python* installation (it is often said that *Python* has "batteries are included") and the rest are relatively easy to find and install.

- The `math` module includes a large number of mathematical functions and constants and will be used very often
- Modules must be import before using their functions/commands (best if done near the beginning of scripts)
- `import math` will import the `math` module
- Include the module name followed by a dot immediately before function/command names from imported modules; i.e. `math.sqrt(5)`

Importing modules

```
>>> import random
>>> random.randint(0, 11)
2
>>> random.choice(range(1, 6))
5

>>> from random import uniform
>>> uniform(-1, 1)
0.5542492833981831
```

14 *Python* Is Good with Math

14.1 Basic Arithmetic

Python accepts common basic arithmetic operators; addition, subtraction, multiplication, division, and exponentiation.

- Addition: `2 + 2`
- Subtraction: `5 - 1`
- Multiplication: `2.5 * 4`
- Division: `7 / 3`
- Exponentiation: `4**2` (Note: the more commonly used `^` is used for a different operation)

Python adheres to the PEMDAS order of operations ...

1. P = Parentheses `( )`
2. E = Exponentiation: `4**2` is 4^2
3. MD = multiplication and division from left to right: `3 * 2 / 3 = 2`
4. AS = addition and subtraction from left to right: `5 - 3 + 1 = 3`

Care should be taken when using multiple, different operators in an expression. Use parentheses where needed to group operations, but excessive overuse of parentheses can make expressions confusing and difficult to troubleshoot when there is an error.

Arithmetic operations

```
>>> 67 + 32 - 13
86
>>> 12 * 6 / 2
36.0
>>> 12 / 2 * 6  # same result as the previous calculation
36.0

>>> 42 - 10/4
39.5
>>> (42 - 10)/4  # parentheses make a difference
8.0

>>> 10 / 5 + 5
7.0
>>> 10 / (5 + 5)  # use parentheses when needed
1.0

>>> 2**3
8
>>> 9**1/2   # the 1/2 is not grouped, so 4**1 will be performed first
4.5
>>> 9**(1/2)
3.0
```

14.2 Special Division Operations

14.2.1 The Division Operation in Mathematics

Division involves four different objects: dividend, divisor, quotient, and remainder

- The dividend is divided up by the divisor
  - Dividend ÷ Divisor
  - Dividend / Divisor
- The quotient is the number of times the divisor completely goes into the dividend
- The remainder is what is left (if anything)
- For example, 7/2 results in a quotient of 3 and remainder of 1

14.2.2 Division in *Python*

- Standard division, i.e `5 / 2`, always yields a floating point value
- Integer division, i.e. `5 // 2`, returns the quotient as an integer
  - Also referred to as floor division
  - Same as performing standard division and only keeping the part left of the decimal
- Modulo division, i.e. `5 % 2`, yields just the remainder
  - Also referred to as remainder division
  - If one value is divisible by another the remainder will be zero
- The built-in function `divmod(x, y)` requires two arguments (the dividend `x` and divisor `y`) and returns the quotient and remainder as a pair of values `(q, r)`

Division operators for 7/2

```
print('Standard division: 7 / 2 =', 7 / 2)
print('Integer division: 7 // 2 =', 7 // 2)
print('Remainder (modulo) division: 7 % 2 =', 7 % 2)
print('Quotient and remainder: divmod(7, 2) =', divmod(7, 2))
```

```
Standard division: 7 / 2 = 3.5
Integer division: 7 // 2 = 3
Remainder (modulo) division: 7 % 2 = 1
Quotient and remainder: divmod(7, 2) = (3, 1)
```

14.3 Elementary Math Functions and Constants

Python is can be easily extended via external modules and libraries. Common mathematical functions and constants beyond basic arimthmetic are provided in the `math` module.

- Must `import` the `math` module before using the functions it contains
- Use `dir(math)` or `print(dir(math))` after importing to see available `math` functions
- Must include the module name when calling functions, i.e. `math.log(2)`

Importing math and listing the available math commands

```
>>> import math
>>> print(dir(math))
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
 ↪  'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb',
 ↪  'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc',
 ↪  'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
 ↪  'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
 ↪  'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma', 'log', 'log10',
 ↪  'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow', 'prod',
 ↪  'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
 ↪  'trunc']
```

- Specific functions can be imported from the `math` (or any other) module (see example below)
- Do not have to include `math.` for functions that are specifically imported

Importing specific math functions

```
>>> from math import cos, sin, tan, pi
>>> cos(pi/3)
0.5000000000000001
```

- Can import all functions from a module by using the `*` wildcard (as shown below)
- Not considered good form nor is it very Pythonic (frowned upon in most cases)
- Please do not use this method unless specified by the instructor

Not a good practice, but it works

```
>>> from math import *
>>> sin(pi/3)
0.8660254037844386
```

14.4 General Math Functions

- `abs(x)` $= |x|$ (absolute value is not part of the `math` module)
- `math.sqrt(x)` $= \sqrt{x}$ (x must not be negative)
- `math.exp(x)` or `math.e**x` $= e^x$
- `math.log(x)` $= \log x$ or $\ln x$ (this is the natural log)
- `math.log(x, b)` $= \log_b x$ (the logrithm of base b)
- `math.log10(x)`$= \log_{10} x$
- `math.factorial(x)` $= x!$ (x must be a positive integer)

General math functions

```
>>> abs(-100)
100
```

```
>>> math.sqrt(5)
2.23606797749979
>>> math.exp(2)
7.38905609893065
>>> math.log(math.exp(2))  # log(e**x) = x
2.0
>>> math.log(25, 5)        # log base 5 of 25
2.0
>>> math.log10(1000)
3.0
>>> math.factorial(5)
120
```

14.5 Math Constants

- `math.pi` $= \pi$
- `math.tau` $= 2\pi$
- `math.e` $= e$
- `math.inf` $= \infty$
- `math.nan` = “Not a number”

14.6 Trigonometric Functions

- The `math` module includes full support for trigonometric functions
- Standard trig functions require/use angle units of radians, not degrees
- Convert angles from degrees to radians using `math.radians(x)` (described later)

Trig functions

```
>>> math.sin(math.pi/4)
0.7071067811865475
>>> math.cos(math.pi/3)
0.5000000000000001
>>> math.tan(math.pi/4)
0.9999999999999999
```

- Inverse (arcus) trig functions return angles in radians, not degrees
- There is a second inverse tangent function named `atan2(y, x)`
  - Accepts two arguments `y, x` instead of one
  - Returns a quadrant-specific angle based on the x and y values
  - Arguments are in y, x order to match y/x, i.e. `atan2(y, x)` equals $\tan^{-1}\left(\frac{y}{x}\right)$

Inverse trig functions

```
>>> math.asin(0.9)
1.1197695149986342
>>> math.acos(0.6)
0.9272952180016123
>>> math.atan(1.0)
0.7853981633974483
>>> math.atan2(-2, -4)
-2.677945044588987
```

- Angles can be converted from degrees to radians or from radians to degrees
  - `math.degrees()` converts an angle from radians to degrees
  - `math.degrees(math.pi/2)` converts $\pi/2$ to degrees
  - `math.radians()` converts an angle from degrees to radians
  - `math.radians(30)` converts $30°$ to radians

Trig functions with angle conversions

```
>>> math.sin(math.radians(30))
0.49999999999999994
>>> math.cos(math.radians(45))
0.7071067811865476
>>> math.degrees(math.atan(1.0))
45.0
>>> math.degrees(math.asin(3/5))
36.86989764584402
```

14.7 Rounding Related Functions

- `round(x, n)`
  - Rounds values towards zero if the decimal part is less than 0.5 and away from zero otherwise
  - Second argument is optional and is used to set the number of decimal places for rounding
  - `round(3.14159, 2)` will round to 2-decimal places, resulting in `3.14`
  - `round(8675309, -2)` will round to the hundreds place, resulting in 8675300
- `math.trunc(x)`
  - Always drops off the decimal portion
  - Leaves an integer value
- `math.ceil(x)`
  - Returns the next integer value towards positive infinity if the argument has a decimal part
- `math.floor(x)`
  - Returns the next integer value towards negative infinity if there is a decimal part

Rounding related functions

```
>>> round(math.e)
3
>>> round(math.e, 3)
2.718
>>> round(8675309, -2)
8675300
>>> math.trunc(math.pi)
3
>>> int(math.pi)  # same as use math.trunc()
3
>>> math.ceil(2 * math.pi)
7
>>> math.ceil(2.0001)
3
>>> math.floor(44.99)
44
```

15 Other Information and Special Commands

- The last calculated result that was not assigned to a name may be accessed using _ (the underscore character)
  - Not normally done within a script or a function definition
  - Acts like the "ans" key on many calculators when working from a *Python* command line
- The `dir()` function returns a list of all current names and modules in memory
- `dir(module)` will return the available commands, functions, and methods in the module named `module`
- *Jupyter* and *iPython* have some *magic* commands
  - `who` is like `dir()` but formatted differently
  - `whos` is like `who` but it also includes types and value information
- Use `help()` to get interactive help
- Use `help(command)` to get help on `command`, i.e. `help(abs)`
  - Press the spacebar to go to the next page of a long help document
  - Press `q` to exit a document
- Use `help("topic")` to get help on `topic` instead of a command or function, for example `help("keywords")`

16 More Code Examples

The code below calculates the magnitude and angle of a vector that whose ending x and y coordinates are provided. It is assumed that the vector's origin is located at $(0, 0)$. Notice that the `x, y` are assigned values at the same time. The magnitude result is rounded to 2-decimal places and the angle is rounded to 1-decimal place in the printed statements.

Vector magnitude and angle

```
import math
x, y = 12, -7    # x = 12, y = -7
magnitude = math.sqrt(x**2 + y**2)
angle = math.degrees(math.atan2(y, x))
print("Given the (x, y) vector components:", x, "and", y)
print(f"  Vector magnitude is: {round(magnitude, 2)}")
print(f"  Vector angle is: {round(angle, 1)} degrees")
```

```
Given the (x, y) vector components: 12 and -7
  Vector magnitude is: 13.89
  Vector angle is: -30.3 degrees
```

Thermistors are temperature measuring devices in whose resistance changes with changes in temperature. One of the methods of calculating the thermistor resistance relative to temperature (temperature relative to thermistor resistance) is with the Beta formula. This formula is shown below; first for calculating the resistance and second for calculating the temperature, depending upon which is known. The values of β, R_o, and T_o are dependent upon the thermistor used. The temperatures in the formula need to be in Kelvin.

$$R_{th} = R_o e^{\beta(1/T - 1/T_o)} \qquad T = \frac{1}{\frac{\ln(R_{th}/R_o)}{\beta} + \frac{1}{T_o}}$$

Thermistor temperature

```
import math
beta = 4450
T_0 = 25             # degrees C
T_0 = T_0 + 273.15   # Kelvin
R_0 = 10000          # ohms
R_th = 8950          # thermistor resistance
T_th = 1/(math.log(R_th/R_0)/beta + 1/T_0)  # in Kelvin
T_th_C = T_th - 273.15
print(f"Thermistor temperature is {round(T_th_C, 1)} deg C")
```

```
Thermistor temperature is 27.2 deg C
```

A university campus has a total of six 90-seat lecture rooms available to give a department-wide final exam such that all enrolled students in the course take the exam at the same time. In order to limit the likelihood of cheating, the department chair has decided only every-other seat will be able to be used; limiting the seating capacity to 45 students per room. The enrollments in the course sections are 25, 32, 28, and 31. The code below uses the remainder division operation and the ceiling function to determine how many rooms will be required and how many available seats will remain.

Final exam rooms needed

```
import math
students = 25 + 32 + 28 + 31
capacity = 90/2
rooms_needed = math.ceil(students/capacity)
available = (45 * rooms_needed) % students
print("There are", students, "students taking the exam")
print(rooms_needed, "rooms are required")
print("There will still be", available, "seats available")
```

```
There are 116 students taking the exam
3 rooms are required
There will still be 19 seats available
```

17 Some Creative Commons Reference Sources for This Material

- *Think Python 2nd Edition*, Allen Downey, chapters 1 and 2
- *The Coder's Apprentice*, Pieter Spronck, chapters 3 and 4
- *A Practical Introduction to Python Programming*, Brian Heinold, chapters 1, 3, 10, and 22
- *Algorithmic Problem Solving with Python*, John Schneider, Shira Broschat, and Jess Dahmen, chapters 2 and 3